GRAME - Computer Music Research Lab.
Technical Report - TR010101

This is a revised version of the previously published paper. It includes a contribution from Shahar Frank who raised a problem with the *fifo-pop* algorithm.
Revised version date: sept. 30 2003.

Optimised Lock-Free FIFO Queue

Dominique Fober, Yann Orlarey, Stéphane Letz
January 2001
Grame - Computer Music Research Laboratory
9, rue du Garet BP 1185 69202 FR - LYON Cedex 01
[fober, orlarey, letz]@grame.fr

Abstract

Concurrent access to shared data in preemptive multi-tasks environment and in multi-processors architecture have been subject to many works. Based on these works, we present a new algorithm to implements lock-free fifo stacks with a minimum constraints on the data structure. Compared to the previous solutions, this algorithm is more simple and more efficient. We'll present its implementation and it's performances.

1 Introduction

Lock-free techniques allow to synchronize concurrent access to shared objects without requiring mutual exclusion. Lock-free objects have several advantages compared to critical sections guarded by locks. If a process is preempted while updating a lock-free object, it will not prevent the other processes from operating on it. Lock-free techniques avoid problems like priority inversion, convoying and deadlock. Moreover, on simple data structures very efficient implementations can be made which are an order of magnitude faster than lock-based implementations.

A lot of works have investigated lock-free concurrent data structures implementations [Anderson & al. 1987, Herlihy 1993, Michael Scott 1996, Valois 1994]. Advantages and limits of these works are discussed in [Michael Scott 1998]. Our implementation is based on [Michael, Scott 1996] but removes the necessary node allocation when enqueing a value, by introducing a simple constraint on the value data type structure.

We present a new lock-free FIFO queue algorithm. It has been initially designed to be part of a multi-tasks, real-time MIDI operating system [Orlarey Lequay 1989, Fober & al 1996] in order to provide an efficient inter-applications communication mechanism. This algorithm is presented in section 2, correctness is discussed in section 3 and experimental results are presented in section 4.

2 Algorithm

Figure 1 presents the data structures used by the algorithm. The FIFO queue is implemented as a linked list of cells with *head* and *tail* pointers. Each pointer have an associated counter, *ocount* and *icount*, wich maintains a unique modification count of operations on *head* and *tail*. A *cell* can be anything provided it starts with a pointer available to link together the cells of the queue.

```
structure cell { next: pointer to next cell, value: data type }
strcuture fifo {
    head: pointer to head cell, ocount: unsigned integer
    tail:   pointer to tail cell, icount: unsigned integer
}
```

Figure 1: data structures

The algorithm relies on an atomic primitive such as *compare-and-swap* which takes as argument the address of a memory location, an expected value and a new value (Figure 2). If the location holds the expected value, it is assigned the new value atomically. A boolean value indicates whether the replacement occurred.

```
compare-and-swap (addr: pointer to a memory location, old, new: expected and new values)
    x <- read (addr)
    if x == old
        write (addr, new)
        return true
    else
        return false
```

Figure 2: atomic compare-and-swap

The *compare-and-swap* primitive was first implemented in hardware in the IBM System 370 architecture [IBM 83]. More recently, it can be found on the Intel i486 [Intel 90] and on the Motorola 68020 [Motorola 86]. A variation of the *compare-and-swap* primitive can also operate in memory on double-words. To differenciate between the two primitives in the following implementation we'll refer to them with:

CAS (mem, old, new) for single word operations
where *mem* is a pointer to a memory location
old and *new* are the expected and the new value

and

CAS2 (mem, old1, old2, new1, new2) for double word operations
where *mem* is a pointer to a memory location
old1, *old2* and *new1*, *new2* are the expected and the new values

As in [Valois 1994] and [Michael, Scott 1996], *head* always points to a dummy cell which is the first cell in the list and *tail* always points to the last or the second last cell in the list. The double-word *compare-and-swap* increments the modification counters to avoid the ABA problem.

The queue consistency is maintained by *cooperative concurrency*: when a process trying to enqueue a cell detects a pending enqueue operation, it initialy tries to complete the pending operation before enqueing the cell. The dequeue operation also ensures that the *tail* pointer does not point to the dequeued cell and if necessary, tries to complete any pending enqueue operation. Figure 3 presents commented pseudo-code for the fifo queue operations.

```
initialize (ff: pointer to fifo, dummy: pointer to dummy cell)
   dummy->next = NULL                   # makes the cell the only cell in the list
   ff->head = ff->tail = dummy          # both head and tail point to the dummy cell


dequeue (ff: pointer to fifo): pointer to cell
D1:    loop                             # try until dequeue is done
D2:        ocount = ff->ocount          # read the head modification count
D3:        icount = ff->icount          # read the tail modification count
D4:        head = ff->head              # read the head cell
D5:        next = head->next            # read the next cell
D6:        if ocount == ff->oc          # ensures that next is a valid pointer
                                        # to avoid failure when reading next value
D7:            if head == ff->head      # is queue empty or tail falling behind ?
D8:                if next == NULL      # is queue empty ?
D9:                    return NULL      # queue is empty: return NULL
D10:               endif
                   # tail is pointing to head in a non empty queue, try to set tail to the next cell
D11:               CAS2 (&ff->tail, head, icount, next, icount+1)
D12:           else if next <> ENDFIFO(ff)      # if we are not competing on the dummy next
D13:               value = next->value          # read the next cell value
                   # try to set head to the next cell
D14:               if CAS2 (&ff->head, head, ocount, next, ocount+1)
D15:                   break            # dequeue done, exit the loop
D16:           endif
D17:       endif
D18:   endloop
D19:   head->value = value              # set the head value to previously read value
D20:   return head                      # dequeue succeed, return head cell
```

```
enqueue (ff: pointer to fifo, cl: pointer to cell)
E1:   cl->next = NULL                        # set the cell next pointer to NULL
E2:   loop                                   # try until enqueue is done
E3:      icount = ff->icount                 # read the tail modification count
E4:      tail = ff->tail                     # read the tail cell
E5:      if CAS (&tail->next, NULL, cl)      # try to link the cell to the tail cell
E6:         break;                           # enqueue is done, exit the loop
E7:      endif
         # tail was not pointing to the last cell, try to set tail to the next cell
E8:      CAS2 (&ff->tail, tail, icount, tail->next, icount+1)
E9:   endloop
E10:  CAS2 (&ff->tail, tail, icount, cl, icount+1)  # enqueue done, try to set tail to the enqueued
cell
```

Figure 3: operations on the lock-free FIFO queue

3 Correctness

3.1 Safety

The main difference with the Michael-Scott algorithm relies on the cells structure constraint, which allows to avoid nodes allocation and release. In fact, the cells memory management is now in charge of the queue clients and may be optimised to the clients requirements but it doesn't introduce any change in the algorithm functionning. Another difference is the modification counts to take account of the ABA problem: they are now associated only to the *head* and *tail* pointers to ensures atomic modifications of these pointers.
Therefore, the properties satisfied by the Michel-Scott algorithm [Michael, Scott 1996] continue to hold ie:

1. the linked list is always connected,
2. cells are only inserted after the last cell in the linked list,
3. cells are only deleted from the beginning of the linked list,
4. head always points to the first node in the linked list,
5. tail always points to a node in the linked list.

3.2 Linearizability

The algorithm is linearizable because each operation takes effect at an atomic specific point [Herlihy, Wing 1987]: E5 for enqueue and D14 for dequeue. Therefore, the queue will never enter any transient state: along any concurrent implementation history, it can only swing between 2 different states S0 and S1 which are acceptable and safe states for the queue:

Assuming a queue in state S0:

- 1) consider an *enqueue* operation : as the queue state is S0, the atomic operation in E5 will succeed and the queue swings to S1 state. Then the atomic operation in E10 is executed: in case of success, the queue swings back to S0, in case of failure a successfull concurrent operation occurs on a S1 state and therefore by 3) and 4), the queue state should be S0.
- 2) consider a *dequeue* operation : if the queue is empty the operation returns in D9 and the state remains unchanged, otherwise the operation atomically executes D14: in case of success, the queue state remains in S0, in case of failure, a concurrent dequeue occured and as it has successfully operated on a S0 queue (by hypothesis) the final state remains also in S0.

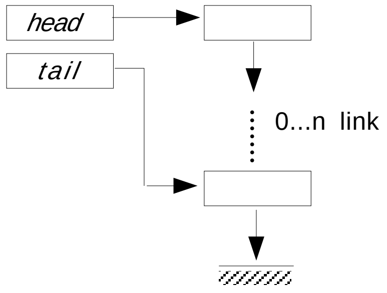


Figure 4: state S0

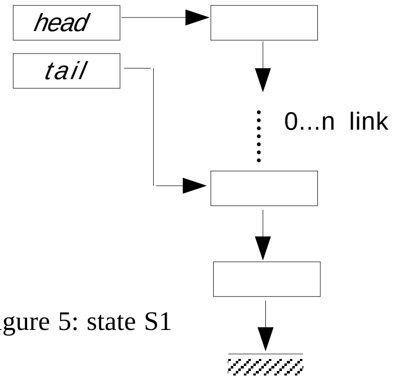


Figure 5: state S1

Assuming a queue in state S1:

- 3) consider an *enqueue* operation : as the queue state is S1, the operation atomically executes E8 and then loops. In case of success, the queue swings to S0 otherwise the operation loops until success ie until the queue is back to S0.
- 4) consider a *dequeue* operation : it is concerned by S1 only if *tail* and *head* points to the same cell which is only possible with a queue containing a single cell linked to the dummy cell. In this case, the operation atomically executes D11 and then loop. In case of success, the queue swings to S0 state. A failure means that a concurrent dequeue or enqueue successfully occured: a successfull dequeue swing the queue to S0 (but it is now empty) and a successfull enqueue too (by 3)

3.3 Liveness

The lock-free algorithm is non-blocking. This is asserted similarly to [Michael, Scott 1996].
Assume a process attempting to operate on the queue:

- the process tries to enqueue a new cell: a failure means that the process is looping thru E8 and then another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell.
- the process tries to dequeue a cell: a failure means that the process is looping thru D11 or D14. A failure in D11 means that another process must have succeeded in completing an enqueue operation or in dequeuing the tail cell. A failure in D14 means that another process must have succeeded in completing a dequeue operation.

4 Performances

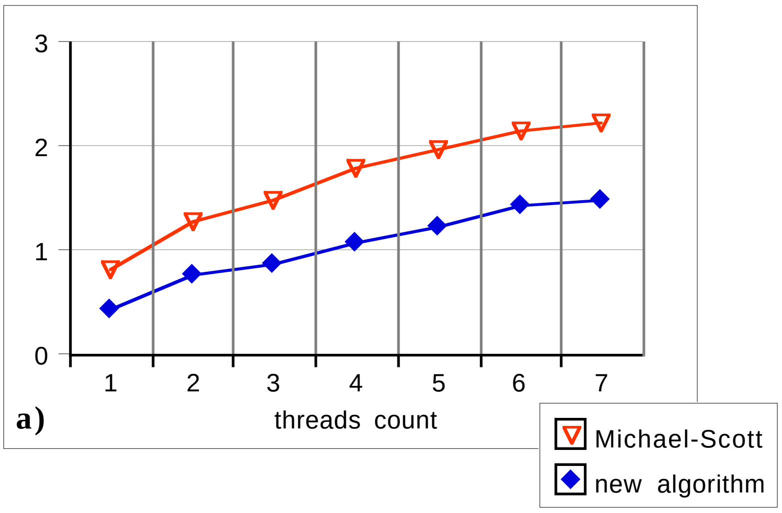
The performances of the new algorithm are compared to a lock-based implementation and to the Michael Scott algorithms (lock-free and 2-locks algorithms) by measuring the time required for 1 to 7 concurrent threads to perform 1 000 000 x 6 concurrent enqueue and dequeue operations on a shared FIFO queue. The tests have been made on a Bi-Celeron 500MHz SMP machine with a 2.2.14 Linux kernel.
The code executed by each thread is shown in Figure 6.

```
long stacktest (long n) {
    cell*       temp[6];
    long        i;
    clock_t     t0, t1;

    t0 = clock();
    while (n--) {
        for (i=0; i<6; i++)
            temp[i] = dequeue(&gstack);
        for (i=0; i<6; i++) enqueue(&gstack, temp[i]);
    }
    t1 = clock();
    return  t1-t0;
}
```

Figure 6: the thread task.

The integrity of the queue was checked after the threads had completed their operations.



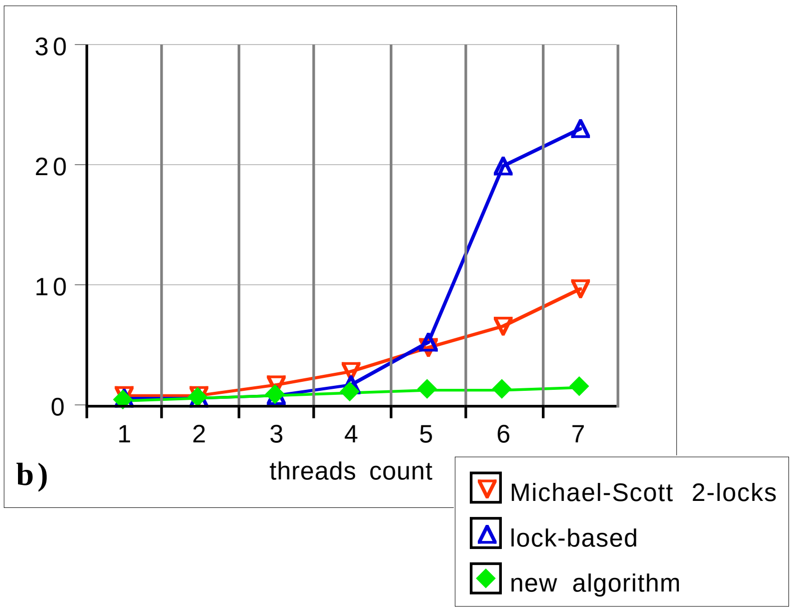


Figure 7: average time (in µs) to perform a dequeue and enqueue operation:
a) compared to Michael-Scott lock-free algorithm
b) compared to a lock-based solution and to the 2-locks Michael-Scott algorithm

In the Michael-Scott implementation, nodes allocation is performed using a statically allocated set of nodes and an index atomically incremented to access the next free node in the table (Figure 8). The node table size prevents multiple node allocation. A node release is implicit and needs no additionnal operation.

```
node_t * new_node() {
    static long index = 0;
    long next, i;
    do {
         i= index;
         next = (i >= MAXNODES) ? 0 : i+1;
    } while (!CAS(&index, i, next));
    return &nodes[next];
}
```

Figure 8: node allocation in Michael Scott implementation

The results show that the new algorithm is more efficient:

- the Michael-Scoot algorithm additionnal cost is between 84% (single thread) and 48% (7 threads)
- the Michael-Scoot 2-locks algorithm additionnal cost is between 15% (two threads) and 553% (7 threads)
- the lock-based solution additionnal cost grows up to 1440% (7 concurrent threads).

5 Aknowledgements

Thanks to Shahar Frank <fesh@exanet.com> who reported the *fifo-pop* problem and for its suggested solution.

References

[Anderson & al. 1987] James H. Anderson, Srikanth Ramamurthy and Kevin Jeffay. “Real-time computing with lock-free shared objects.” ACM Transactions on Computer Systems Vol. 15, No. 2, May 1997, pp. 134 - 165

[Herlihy, Wing 1987] M. P. Herlihy, J. M. Wing. “Axioms for concurrent objects.” In Proceedings of the 14th ACM Symposium on Principles of Programmmg Languages,Jan. 1987, pp. 13-26.

[Herlihy, Wing 1990] M. P. Herlihy, J. M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects.” ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, July 1990, pp. 463-492.

[Herlihy 1993] M. P. Herlihy. “A methodology for implementing highly concurrent data objects.” ACM Trans. Program. Lang. Syst. 1993, Vol. 15, No.5, pp. 745–770.

[IBM 83] International Business Machines Corp. “System / 370 Principles of Operation” 1983

[Intel 90] Intel Corporation. “i486 Processor Programmer’s reference Manual” Intel, Santa Clara, CA, 1990

[Michael, Scott 1996] M. M. Michael and M. L. Scott. “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.” 15th ACM Symp. on Principles of Distributed Computing (PODC), May 1996. pp. 267 - 275

[Michael, Scott 1998] M. M. Michael and M. L. Scott. “Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors.” Journal of Parallel and Distributed Computing, 1998, pp. 1-26.

[Motorola 86] Motorola. “MC68020 32-Bit Microprocessor User’s Manual” Prentice-Hall, 2nd edition, 1986

[Orlarey, Lequay 1989] Y. Orlarey, H. Lequay. “MidiShare : a Real Time multi-tasks software module for Midi applications” Proceedings of the International Computer Music Conference 1989, Computer Music Association, San Francisco, pp.234-237

[Fober & al 1996] D. Fober, S. Letz, Y. Orlarey. “Recent developments of MidiShare” - Proceedings of the International Computer Music Conference 1996, International Computer Music Association, San Francisco, p.40-42

[Valois 1994] John D. Valois. “Implementing Lock-Free Queues.” Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, October 1994, pp. 64-69

[Valois 1995] John D. Valois. “Lock-Free Data Structures.” Ph.D. Thesis, Rensselaer Polytechnic Institute, new York, May 199